Trabajo final de máster

Exploración coordinada entre robot móvil terrestre y dron aéreo

Robótica móvil

Víctor Valseca Martínez

Universitat Jaume I

Máster Universitario en Sistemas Inteligentes

Tutor

Prof. Enric Cervera Mateu

Fecha de la defensa

--

A todos los que ya no están, a los que construyeron este presente y a los que ayudarán a resolver el futuro

Agradecimientos

"Es de bien nacido ser agradecido". Con este refrán en mente doy paso a los agradecimientos. Al menos para mí dar las gracias a las personas que me han ayudado de alguna forma es una manera de devolver al menos un poquito de lo que han dado, su tiempo y dedicación, pues ninguno de los que aquí figuran han gastado su tiempo a desgana conmigo y merecen toda la atención que les pueda dar.

En primer lugar, quiero agradecer a mi tutor todo el tiempo que ha dedicado a ayudarme, sin su inestimable ayuda no hubiera podido realizar un trabajo en condiciones. Además, quisiera agradecer a los demás profesores del máster su profesionalidad y desempeño enseñando. Da gusto aprender así.

Para seguir, me gustaría agradecer a mi familia todo el apoyo que me han brindado desde la distancia. Han sido y son un pilar fundamental en mi vida. En especial agradecer a mis abuelos y a mis padres todo el apoyo en los momentos difíciles.

Agradecer a mi hermano el no haberme dejado solo nunca y haberme acompañado en el trascurso del máster desde la distancia.

Para mis amigos también tengo unas palabras de agradecimiento. En especial a Tino, a Pablo y a Carlos, por estar presente en las buenas y en las malas, por dedicarle noches y días a aguantarme.

Y hablando de aguantar, tengo que agradecer a Pan de Pueblo el estar ahí y mejorar mi vida.

A Legacy como siempre le tengo que agradecer los buenos momentos que me da. Y en especial, a Kusari(Daniel), Maties y Rai(Rosa), que dedicaron parte de su tiempo a hacerme compañía.

Para la cabra cósmica más bonita de Murcia. atte.: El enano que "cazó" tu corazón.

Y para Ruti, el escudo de Legacy.

Resumen

En este documento, que comprende al trabajo final de Máster, se presenta una solución para el movimiento coordinado de dos robots, un UGV (Unmanned Ground Vehicle) que estará teleoperado y un UAV (Unmanned Air Vehicle) que seguirá al UGV automáticamente.

El robot móvil Turtlebot [2] hará la función del UGV, y el dron Parrot Bebop 2 [3] realizará el papel del UAV.

El seguimiento del UAV se realizará por medio de una etiqueta localizada en el mismo dron y vista por una cámara USB que estará situada en la parte superior del Turtlebot.

Mediante ROS (Robot Operating System) [4], se utilizarán diversos paquetes para ejecutar los controladores de dichos robots, y no solo eso, si no que se ejecutarán paquetes que posibilitarán la localización de la etiqueta mediante las imágenes recibidas por la cámara USB, y de esta forma saber la posición del dron con respecto al Turtlebot.

Para posibilitar que el dron se mueva al mismo tiempo que el Turtlebot y trate de mantener una posición y orientación fija encima del Turtlebot, se ha programado el nodo tracking_nodev2, el cual es un programa sencillo que posibilita el desplazamiento conjunto.

Para acabar, en este trabajo se enuncian algunas posibles mejoras para tratar de enriquecer el trabajo.

Abstract

This document, which includes the final work of the Master, presents a solution for the cooperative movement of two robots, an UGV (Unmanned Ground Vehicle) which is teleoperated and an UAV (Unmanned Air Vehicle) that follow the UGV automatically.

The mobile robot Turtlebot [2] will perform the function of the UGV and the drone Parrot Bebop 2 [3] will perform the role of the UAV.

The tracking of the UAV will be done by a tag located in the same drone and seen by a USB camera that will be located in the top of the Turtlebot.

By means of ROS (Robot Operating System) [4], several packages will be used to execute the controllers of these robots, and not only that, packages will be executed that will allow the location of the tag by means of the images received by the USB camera, and of this form to know the position of the drone with respect to the Turtlebot.

To allow the drone to move at the same time as the Turtlebot and try to maintain a fixed position and orientation on top of the Turtlebot, the tracking_nodev2 has been programmed, which is a simple program that allows the joint movement.

To finish, in this work some possible improvements are enunciated to try to enrich the work.

Índice general

Índice de imágenes

Índice de tablas

1. Introducción

A medida que la tecnología avanza, también van naciendo nuevas líneas de desarrollo. Una de esas líneas contiene a los robots autónomos, los cuales son capaces de realizar tareas que los humanos pueden desarrollar y otras tantas que resultan imposibles o tienen un difícil acceso, como puede ser el caso de la inspección de redes eléctricas en entornos naturales, la exploración de las profundidades marinas o la exploración espacial.

Sin embargo, bien es sabido que el ser humano es un ser social que necesita de la ayuda de otros, para poder realizar tareas que a priori pueden ser difíciles para un solo individuo. Haciendo que dichas tareas disminuyan su dificultad hasta ser considerada un trámite menor. Extrapolando tal caso hacia los robots autónomos, nos encontramos con las mismas ventajas de la cooperación humana pero trasladado al plano robótico, el cual es capaz de desempeñar acciones que un grupo de seres humanos no podría.

Aunque este campo aún no está del todo desarrollado, nos podemos encontrar muchas soluciones para los distintos problemas que se tratan de resolver. Un ejemplo de ello es mostrado en el artículo "*Bird's Eye View: Cooperative exploration with UAV and UGV*" [1]. La motivación de dicho artículo era realizar una aplicación de búsqueda y rescate en construcciones dañadas.

En el mismo proponen una solución al mapeado bidimensional y tridimensional del área [1] en el cual poder localizar los objetivos, a modo de reconocimiento previo, pues en un rescate es indispensable primero conocer la situación del entorno para poder operar adecuadamente.

Tal y como dice el nombre del artículo, utiliza un equipo de dos robots móviles, uno aéreo [3] y otro terrestre[2] para realizar la exploración conjunta del entorno. El problema radica en la localización de la posición de un robot con respecto al otro. Hecho que solucionan mediante la asignación de etiquetas AR a uno de los dos robots, con el objetivo de localizar la posición exacta del vehículo etiquetado. Toda la comunicación entre nodos y ejecutables que se han utilizado tienen como base ROS [4].

El artículo [1] nos describe cuáles son sus componentes principales, Parrot Bebop 2 [3], Turtlebot 2 [2] y ROS [4], los cuales utilizará para realizar el mapeado en tres dimensiones del entorno. Para realizar el trabajo conjunto, utilizará un paquete que identifica las etiquetas Alvar, el paquete ar_track_alvar [7]. A partir del cual se podrá saber cuál es la posición y orientación de la etiqueta captada por la cámara, que estará localizada en la parte superior del Turtlebot. Sabiendo la posición en todo momento del dron con respecto al Turtlebot, se puede obtener la odometría del dron sabiendo la odometría del Turtlebot.

Para obtener un mapa del entorno, el Turtlebot utiliza el sensor Orbbec Astra Pro Sensor, el cual es similar a un sensor Kinect y realiza el mapa en dos dimensiones haciendo uso de un .launch (gmapping_demo.launch) que viene incluido en el propio Turtlebot sin necesidad de descarga o instalación previa.

Por otro lado, el dron contribuye a la elaboración del mapa tridimensional haciendo uso de ORBSLAM2 [8], el cual utiliza las imágenes captadas por la cámara del dron para realizar una correspondencia entre imágenes y hallar la trayectoria del dron. Para posteriormente, gracias a los puntos tridimensionales que ha hallado ORBSLAM2 y el mapa realizado por el Turtlebot elaborar un mapa de tres dimensiones que contiene tanto información de un robot como de otro.

Este trabajo se va a basar en principalmente en el documento [1] para resolver el problema de seguimiento por visión de un robot no tripulado, que como en el caso del artículo será el dron, a otro que será teleoperado que en este caso, emulando al trabajo expuesto en el artículo nuevamente, será un Turtlebot.

Este documento está compuesto por una serie de secciones que tratarán de explicar cómo se ha desarrollado el trabajo:

La sección 1 corresponde a la introducción al proyecto.

La sección 2, corresponde a la planificación, metodología del trabajo y objetivos a cumplir.

La sección 3, se describirán brevemente los componentes, tanto de hardware como de software, que se han utilizado para la consecución del trabajo expuesto en este documento. Además se realizarán algunas recomendaciones sobre alguno de éstos componentes.

La sección 4, tratará sobre cómo se ha desarrollado el proyecto desde el inicio, que es investigar a fondo el artículo hasta la programación del nodo que realiza la cooperación UGV-UAV. Además se detalla cómo se han realizado los diversos estadios que componen la interacción entre un robot y otro.

En la sección 5 se comentarán las conclusiones generadas de esta investigación y los resultados obtenidos.

La sección 6 contendrá el trabajo a desarrollar en un futuro si se pretende mejorar la aplicación desarrollada, que aunque básica, puede establecer un punto de apoyo para tareas más difíciles de desarrollar . Además se nombrarán diversos paquetes que pueden ser de solución para lo que se explicará en esta sección.

2. Planificación

2.1. Objetivos

Como se ha mencionado en el apartado anterior, los principales objetivos son:

- Hacer que el Parrot siga al Turtlebot, el cual está siendo teleoperado.
- Como extensión, tratar de replicar dentro de lo posible todo lo expuesto en el artículo [1]. El cual se encarga de establecer un desplazamiento conjunto con el fin de realizar un mapa tridimensional elaborado a partir de las correspondencias de los mapas hallados por el dron y el Turtlebot.
- Mejorar o extender las funcionalidades que el artículo [1] nos presenta.

2.2. Metodología

Los métodos que se utilizarán para la consecución del trabajo son los siguientes:

- Estudio en profundidad de la información ofrecida en el artículo.
- Estudio de los diferentes componentes que conforman el proyecto (paquetes de ROS, aplicaciones de odometría visual, aplicaciones de localización, etc.).
- Análisis y prueba de los componentes.
- Enlace de los componentes y evaluación de los resultados obtenidos.
- Estudio sobre las posibles adiciones que se pudieran incluir en el artículo, y si es posible realizarlo.

2.3. Planificación

Tarea	Horas planificadas	Horas realizadas	Objetivo
Estudio del artículo	20 horas	30 horas	Entender el artículo en su totalidad
Estudio de los componentes	70 horas	50 horas	Entender los componentes utilizados, al menos teóricamente
Prueba de los componentes	70 horas	110 horas	Comprensión de los componentes a través de la prueba práctica
Enlace de los componentes	60 horas	20 horas	Comprendidos los componentes, tanto teóricamente como experimentalmente, queda enlazarlos
Posibles añadidos y diferentes pruebas a realizar	30 horas	20 horas	Evaluación sobre las posibles mejoras que pudiera tener
Elaboración de la memoria	40 horas	60 horas	Constatar por escrito todo lo realizado
Realización de la presentación del trabajo	10 horas	10 horas	Además de presentar el trabajo ante un tribunal que evalúe el mismo, también es necesario preparar la misma

Tabla 1 Planificación

3. Componentes

A continuación se describirán brevemente aquellos componentes esenciales que se han utilizado.

3.1. Hardware

En esta sección se verán aquellos componentes que conforman el sistema para realizar la cooperación entre el dron y el robot móvil terrestre. Además de los dispositivos que se verán a continuación es necesario resaltar que se ha tenido a disposición dos ordenadores que han hecho posible la comunicación entre ambos mediante los componentes de software que se explicarán posteriormente. Uno de estos ordenadores estaba situado en el Turtlebot y el otro hacía de centro de operaciones desde donde se ejecutaban todos los comandos. Este último, tenía instalado ROS Indigo y ejecutaba los nodos necesarios para la cooperación.

3.1.1. Parrot Bebop 2

En primer lugar tenemos el dron, el Parrot Bebop 2, el cual, con un peso de 500 gramos y una autonomía de 25 minutos, es un dron de ocio de última generación. Sus prestaciones le permiten volar, grabar y tomar fotos al mismo tiempo, tanto en interiores como en exteriores.

El Parrot Bebop 2 te permite grabar en Full HD 1080p. Incluso con poca luminosidad se puede disfrutar de una nitidez considerable gracias a la lente gran angular de 14 megapíxeles que porta. Gracias a que la cámara rota según la orientación que el dron posea, se pueden obtener secuencias de imágenes estables sin vibraciones o perturbaciones que dificulten la captura de éstas. Además, posee una cámara en la parte inferior que apunta hacia abajo que puede ser utilizada para diversas tareas.

Puede alcanzar una velocidad máxima de 60 km/h en horizontal y 21 km/h en vertical. Puede alcanzar su velocidad máxima en 14 segundos y resistir vientos de frente de hasta 60 km/h y además, tiene un tiempo de frenada 4 segundos.

Parrot Bebop 2, puede ser teleoperado mediante un smartphone. Responde a los movimientos del dispositivo móvil gracias al acelerómetro y a los mandos táctiles de la interfaz. Para simplificar el pilotaje, el despegue y el aterrizaje se realizan automáticamente con un simple botón. Además también existe un driver que puede controlarlo mediante ROS que posibilita hacer todo lo anterior y que se describirá más adelante en la sección de Software.

Gracias a la aplicación que posee puede entrar en modo *follow me,* con el cual gracias a la localización GPS del smartphone podrá seguir a la persona que lo esté controlando sin necesidad de estar mandando órdenes a cada instante.

Figura 1 Parrot Bebop 2

Especificaciones técnicas:

Imagen	• Cámara de 14 megapíxeles con lente gran angular • Formato de la foto: RAW, JPEG, DNG
Especificaciones	• Hélices flexibles que se bloquean en caso de contacto • LED trasero visible a larga distancia • GPS integrado para el control de vuelo en altitud y retorno automático al punto de despegue • Procesador dual-core con GPU quad-core • 8 GB de memoria de almacenamiento flash • Fácil de manejar con la aplicación FreeFlight Pro disponible tanto para iOS como para Android
Estructura	• 4 motores sin escobillas Outrunner • PA12 estructura reforzada de fibra de vidrio (20%) y Grilamid (casco) • Peso: 500 g • Dimensiones: 38 x 33 x 9 cm
Video	• Resolución de vídeo: 1920 x 1080p (30 fps) • Codificación de vídeo: H264 • Estabilización de vídeo: sistema digital de 3 ejes
Autonomía	• 25 minutos de vuelo con la batería 2700 mAh
Conectividad	• Wi-Fi 802.11a/b/g/n/ac • Antena Wi-fi: MIMO de banda dual con 2 antenas dipolo dual 2.4 y 5 GHz • Transmisión de potencia: ≤ 21 dBm • Alcance de la señal: 300 m

Tabla 2 Especificaciones técnicas Parrot Bebop 2

3.1.2. Turtlebot 2

Nuestro UGV será un Turtlebot 2 el cual es el robot de código abierto más barato para educación e investigación. Está equipado con una base robot Kobuki, un portátil dual-core, el Orbbec Astra Pro Sensor (similar al Kinect) y un giróscopo. Todos los componentes se han integrado para ofrecer una plataforma de desarrollo lista para usar. Para utilizar este robot será necesario utilizar ROS, lo cual es una ventaja, ya que es una comunidad de desarrolladores de código abierto ideal para comenzar a iniciarse en el mundo de la robótica, y de la cual hablaremos más adelante.

Turtlebot fue diseñado en colaboración con los fundadores originales de ROS, Willow Garage. Desde entonces, ROS se ha convertido rápidamente en la plataforma de software de referencia para los robots de todo el mundo. El Turtlebot no sólo se integrará a la perfección con tus robots existentes impulsados por ROS, sino que también ofrece una plataforma asequible para empezar a aprender con ROS.

Por tanto, ROS viene pre-instalado y configurado en el portátil. Turtlebot viene totalmente montado, integrado y testado. Está listo para operaciones básicas una vez salga de la caja.

Otra de las características más importantes de Turtlebot es la base de datos de tutoriales online, videos y código de ejemplo que permiten aprender ROS de manera rápida y sencilla.

Figura 2 Turtlebot 2

Especificaciones:

Tamaño y peso	
Dimensiones	354 x 354 x 420 mm
Peso	6.3 kg (13.9 lbs)
Máxima carga	5 kg (11 lbs)
Velocidad y desempeño	
Máxima velocidad	0.65 m/s (25.6 in/s)
Distancia entre obstáculos	15 mm (0.6 in)
Drivers y aplicaciones	ROS

Tabla 3 Especificaciones técnicas Turtlebot 2

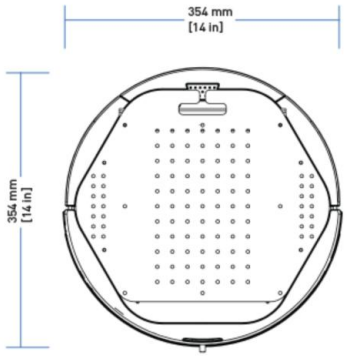

Figura 3 Parte superior Turtlebot 2

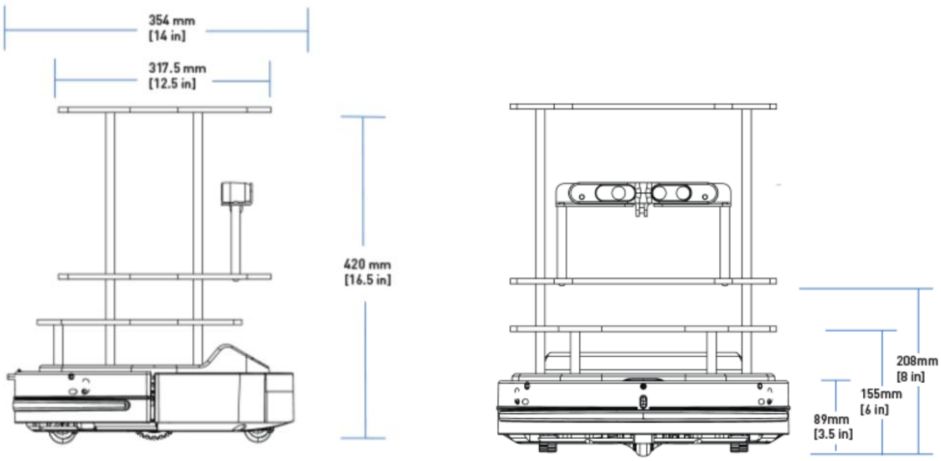

Figura 4 Perfil y Frontal Turtlebot 2

3.2. **Software**

A continuación, veremos todos aquellos componentes de software mencionados con anterioridad, más aquellos que posibilitan realizar la cooperación entre UGV y UAV.

3.2.1.**Ubuntu**

Ubuntu es el sistema operativo que se ha usado para alojar ROS. Es un sistema operativo basado en GNU/Linux que se distribuye como software libre. Es un sistema modular orientado a proporcionar facilidad de uso. El primer lanzamiento del sistema operativo Ubuntu, fue el 20 de octubre de 2004. La versión más reciente y confiable es Ubuntu 18.04.1 (Bionic Beaver) lanzada en el año 2018. No obstante, la versión que se ha utilizado para desarrollar el trabajo es la versión de Ubuntu 14.04 LTS Trusty Tahr.

Las principales razones por las que se ha usado esta versión de Ubuntu, es porque ROS (Sistema Operativo del Robot) solo está disponible para dicha plataforma, además de que la mayoría de paquetes clave a partir de los cuales se ha estado investigando en este proyecto, están realizados en ROS índigo, el cual solo es válido para Ubuntu 14.04.

Una razón adicional por la cual es interesante usar Ubuntu, es porque es un sistema operativo de código abierto y cuenta con una comunidad de desarrolladores bastante extensa, que mejoran dicho sistema de manera continua.

3.2.2. ROS

ROS (Robot Operating System) [4] es un conjunto de librerías y herramientas que permiten crear aplicaciones para robots. ROS no es un sistema operativo en sí, sin embargo, provee de los servicios estándar de éste tales como abstracción de hardware, control de dispositivos de bajo nivel, implementación de funcionalidades usadas habitualmente, paso de mensajes entre procesos y mantenimiento de paquetes. También ofrece herramientas y librerías para escribir y ejecutar código mediante múltiples computadoras.

El ámbito de ROS abarca una red *peer-to-peer* (de igual a igual) de nodos, que pueden estar distribuidos entre varias computadoras y que se comunican libremente usando la infraestructura de comunicaciones de ROS.

La meta de ROS no es ser el entorno con el mayor número de características. De hecho el principal fin es apoyar la reutilización de código en la búsqueda y desarrollo de aplicaciones. ROS es un marco de procesos distribuido que posibilita su realización para ser diseñadas de forma aislada y desacopladas en el tiempo de ejecución. Estos procesos pueden ser agrupados en *Packages* (paquetes) y *Stacks* (Pilas), los cuales son más fáciles de distribuir. ROS también ampara un sistema federado de *Repositories* (Depósitos) que posibilita la colaboración entre grupos.

Para poder lograr el fin último de compartir y colaborar, ROS posee diferentes características que se van a enumerar a continuación:

- Liviano: ROS está diseñado para ser lo más ligero posible, por lo que el código escrito para ROS puede ser usado con otros tipos de *softwares* de robots.
- Librerías *ROS-agnostic*: el modelo de desarrollo preferido es escribir librerías *ROS-agnostic* con interfaces funcionales.
- Independencia de lenguajes: el entorno de ROS es relativamente fácil de implementar en cualquier lenguaje de programación. Tenemos a nuestra disposición librerías en Python, C++ y Lisp. Aunque también existen librerías experimentales en Java y Lua. Lo que queremos decir con ello, es que nuestros nodos o programas podrán estar programados en los distintos lenguajes anteriormente nombrados, sin posibilidad relativa de conflicto entre ellos.
- Fácil de probar gracias a que posee un entorno de pruebas llamado "rostest", la cual hace más fácil la comprobación y reparación de los errores que se pudieran cometer.
- Ajuste: ROS es apropiado para sistemas con tiempos de ejecución extensos y para un número amplio de procesos.

Sistema de archivos

Los componentes del sistema de archivos que abarcan la mayoría de los recursos de ROS son los siguientes:

- Paquetes(*Packages*): es la unidad de organización de *software* del código de ROS. Cada paquete puede contener librerías, nodos, *scripts*, archivos de configuración, etc. En definitiva, cualquier documento que pueda ser útil organizado todo en un mismo directorio.
- Metapaquetes (*Metapackages*): Son paquetes especializados cuyo propósito es representar a un grupo relacionado de paquetes.
- Manifiesto de paquetes (*Package Manifests*): es la descripción de un paquete. Su propósito es definir las dependencias del paquete, los tipos de mensajes y guardar información acerca del mantenedor del mismo, la versión, licencia, etc.
- Depósitos (*Repositories*): es una colección de paquetes que comparte un mismo sistema de control de versiones y pueden actualizarse mediante una herramienta que proporciona el entorno de ROS.
- Tipos de mensajes (msg): descripciones de mensajes, definen la estructura de datos para los mensajes mandados en ROS.
- Tipos de servicios (srv): descripciones de servicios, definen las peticiones y las respuestas de la estructura de datos para los servicios en ROS.

Grafos de computación

El grafo de computación es una red de igual a igual (*peer-to-peer*) de los procesos de ROS que están tratando datos de manera conjunta. Las ideas básicas del grafo de computación de ROS son los nodos, Maestro, Servidor de parámetros, mensajes, servicios, *topics* (temas), y *bags* (bolsas), todo ello provee de información al grafo en diferentes aspectos.

Estos conceptos están implementados en el *repository* "`ros_comm`":

- Nodos: es un ejecutable que usa ROS para comunicarse con otros nodos por medio de los *topics* o servicios. ROS está diseñado para ser modular por lo que un sistema de control robótico tendrá normalmente varios nodos. Un nodo de ROS estará escrito normalmente en C++ o Python, utilizando las librerías roscpp o rospy.
- Maestro: el Maestro de ROS provee de un nombre de registro y cuida del resto del grafo de computación. Sin el Maestro, los nodos no podrían encontrarse los unos con los otros, cambiar mensajes o invocar servicios.
- Servidor de parámetros: permite guardar y recuperar los datos de una localización específica.
- Mensajes: los nodos se comunican entre ellos mediante el uso de mensajes. Un mensaje es una estructura simple de datos que contiene campos determinados (entero, lógico, float, etc).
- *Topics*: los mensajes entre nodos se mandan mediante una vía de transporte con la semántica de publicación/suscripción. Un nodo envía un mensaje por medio de una publicación a un *topic* dado. El *topic* es un nombre que es usado para identificar el contenido del mensaje. El nodo que está interesado en cierto tipo de dato se suscribirá al *topic* apropiado. Puede haber múltiples nodos publicadores y suscriptores para un solo *topic* y solo un nodos que pueda publicar/suscribirse a varios *topics*.
- Servicios: se utiliza para interacciones de petición/respuesta. Un nodos ofrece un servicio bajo un cierto nombre, y otro nodo solicita dicho servicio enviando una petición y esperando una respuesta. Tal es el caso como se hablará más tarde de la calibración de la cámara.
- *Bags*: es un método para guardar y reproducir datos de mensaje de ROS. Es de utilidad para el desarrollo y las pruebas de algoritmos. Ya que a veces puede ser difícil obtener información útil de algunos sistemas.

3.2.3. ar_track_alvar

Este paquete es un una envoltura ROS para Alvar [13], una librería de seguimiento de etiquetas AR de código abierto que tiene varias funcionalidades:

- Generar etiquetas AR de distinto tamaño, resolución y datos intrínsecos.
- Identificación y seguimiento de la posición de etiquetas individuales AR.
- Identificación y seguimiento de la posición del conjunto o bloque de múltiples etiquetas. Esto permite una estimación de la posición más robusta a las oclusiones, y seguimiento de objetos de varias caras.

Alvar cuenta con umbrales adaptables para manejar una variedad de condiciones de iluminación, seguimiento basado en el flujo óptico para una estimación más estable de la postura, y un método mejorado de identificación de etiquetas que no se ralentiza significativamente a medida que aumenta el número de etiquetas.

3.2.4. bebop_autonomy

Es un controlador de ROS para los drones Parrot Bebop 1.0 y 2.0, basado en la aplicación oficial de Parrot, ARDroneSDK3.

Ofrece las mismas posibilidades que podría ofrecer la aplicación de Android o iOS, solo que adaptado a ROS de forma que se puedan añadir algoritmos que permitan realizar más tareas de las que pudiera desempeñar la aplicación.

Este paquete puede iniciarse como un nodo o como un nodelet.

Para iniciarlo como nodo, el mismo se llama `bebop_driver_node` y está en el paquete `bebop_driver`. Es recomendable iniciarlo en su propio espacio de nombres y con la configuración por defecto. El paquete del driver viene con un archivo launch, localizado en `bebop_driver/launch/bebop_node.launch`.

Para iniciarlo como nodelet, primero se necesita iniciar un manejador de nodelet, y luego cargar el driver nodelet en él, y con otros nodelets que necesiten comunicarse con el driver. El archivo launch `bebop_tools/launch/bebop_nodelet_iv.launch` realiza todos estos pasos.

Como todo driver que puede ejecutarse como un nodo de ROS, este posee unos *topics* donde es posible mandar comandos al dron:

- Para hacer despegar al dron, publicaremos en `bebop/takeoff` un mensaje `std_msgs/Empty`
- Para hacer aterrizar al dron, publicaremos en `bebop/land` un mensaje `std_msgs/Empty`
- Para hacer un reset al dron, publicaremos en `bebop/reset` un mensaje `std_msgs/Empty`
- Para pilotar el dron mandaremos un comando Twist a `/bebop/cmd_vel`
- Para mover la cámara virtual, enviaremos un comando Twist a `bebop/camera_control`

Hay que mencionar, que para trabajar con la cámara es necesario hallar los parámetros intrínsecos de ésta, los cuales serán hallados por ejemplo con el paquete `camera_calibration` [9], el cual posee un servicio (`set_camera_info`) que guarda automáticamente la calibración de la cámara en un directorio específico donde el driver pueda recogerlo y publicarlo en `bebop/camera_info`. A la información de la cámara le acompañan las imágenes, las cuales están siendo publicadas en `bebop/image_raw`.

Los parámetros necesarios para configurar el driver del dron son los siguientes:

- `bebop_ip`: Establece la dirección IP del bebop. Por defecto utiliza `192.168.42.1`.
- `reset_settings`: Dándole el valor `true`, se hará un reset de todas las configuraciones del bebop a los valores de fábrica. Por defecto el valor es `false`.
- `sync_time`: Poniendo este valor a `true` se sincronizará el tiempo del dron con el de sistema de ROS, Por defecto es `false`.
- `camera_info_url`: Establece la localización del archivo de calibración de la cámara.
- `cmd_vel_timeout`: Establece el tiempo de espera para los comandos que se van recibiendo y no se ejecutan al momento. Por defecto es 0.1 segundos. Si no se manda ningún comando en este periodo, el dron permanecerá parado.
- `camera_frame_id`: Establece el marco de la cámara y los mensajes de las imágenes. Por defecto es `camera_optical`.

4. Desarrollo del proyecto

A continuación, se explicará el método de tracking visual y el ensayo realizado para probar el nodo programado que posibilite la cooperación entre el Turtlebot y el Parrot.

Se han explorado varias vías para poder llegar a un resultado satisfactorio que cumpliese el objetivo propuesto, hacer que el dron siguiese al Turtlebot, el cual está siendo teleoperado. Con la meta de utilizar este movimiento conjunto para que ambos robots realicen tareas más complejas y colaboren entre sí.

4.1. Descripción del paquete `ar_track_alvar`

A continuación, se explicarán las razones de la inclusión del paquete `ar_track_alvar` en el proyecto. La principal razón de por qué se incluyó este paquete y no otro que desempeñase un papel similar, es que sirve para identificar y rastrear las poses de una o varias etiquetas AR. La segunda razón es sencilla, el artículo [1] en el que se ha basado el trabajo, lo utiliza para saber la posición del dron con respecto al Turtlebot [2].

Para operar con este paquete, al disponer de una cámara USB y no de una Kinect, se ha usado el nodo `individualMarkersNoKinect`, el cual toma los siguientes parámetros de la línea de argumentos:

- `marker_size` (double): la anchura en centímetros de un lado del cuadrado que conforma la etiqueta
- `max_new_marker_error` (double): Un umbral que determina cuándo pueden detectarse nuevos marcadores en condiciones de incertidumbre.
- `max_track_error` (double): Un umbral que determina cuánto error de seguimiento puede observarse antes de que se considere que una marca ha desaparecido.
- `camera_image` (string): Nombre del *topic* del cual recibe las imágenes para detectar las etiquetas AR que se encuentren en ellas.
- `camera_info` (string): Nombre del *topic* que contiene los parámetros de calibración de la cámara para que la imagen pueda ser rectificada.
- `output_frame` (string): Nombre del *topic* que publica las coordenadas cartesianas a las que la etiqueta es relativa.

Para utilizar este paquete en el proyecto, se programó un archivo .launch, el cual daba todos los valores necesarios a los parámetros anteriormente mencionados.

De lo anterior podemos saber, que este paquete tiene como *topics* subscriptores tanto la imagen de la cámara que vaya a identificar la etiqueta o etiquetas y la calibración de ésta.

Sin embargo, de este paquete se obtiene un resultado, en forma de dos *topics*:

- `visualization_marker` (visualization_msgs/Marker): El cual es un mensaje rviz que cuando se suscribe mostrará un bloque cuadrado de color en la ubicación de cada etiqueta AR identificada, y también superpondrá estos bloques en una imagen de cámara.
- `ar_pose_marker` (ar_track_alvar/AlvarMarkers): Es un mensaje que contiene la pose de la etiqueta observada.

Para saber cómo funcionaba este paquete y cómo tenía dispuestos los ejes, se realizaron múltiples pruebas, las cuales consistían en probar la dirección y sentido de los ejes, así como

su robustez ante cambios de luminosidad y orientación. Estas pruebas se realizaron con el objetivo principal de elaborar el programa de cooperación que se describirá más adelante.

Para saber cómo mandar los comandos de velocidad al dron, fue necesario realizar varias pruebas para determinar la disposición del los ejes en la etiqueta.

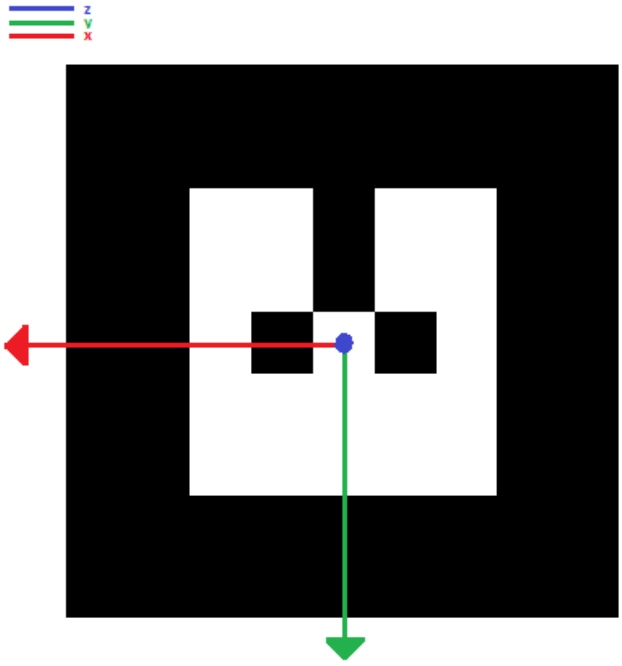

Figura 5 Etiqueta Alvar con ejes

En la figura 5, el sentido de los ejes señala el área donde la posición de la etiqueta se hace positiva. Hay que resaltar, que el eje z apunta hacia afuera del papel, no hacia adentro.

4.2. Descripción del ensayo

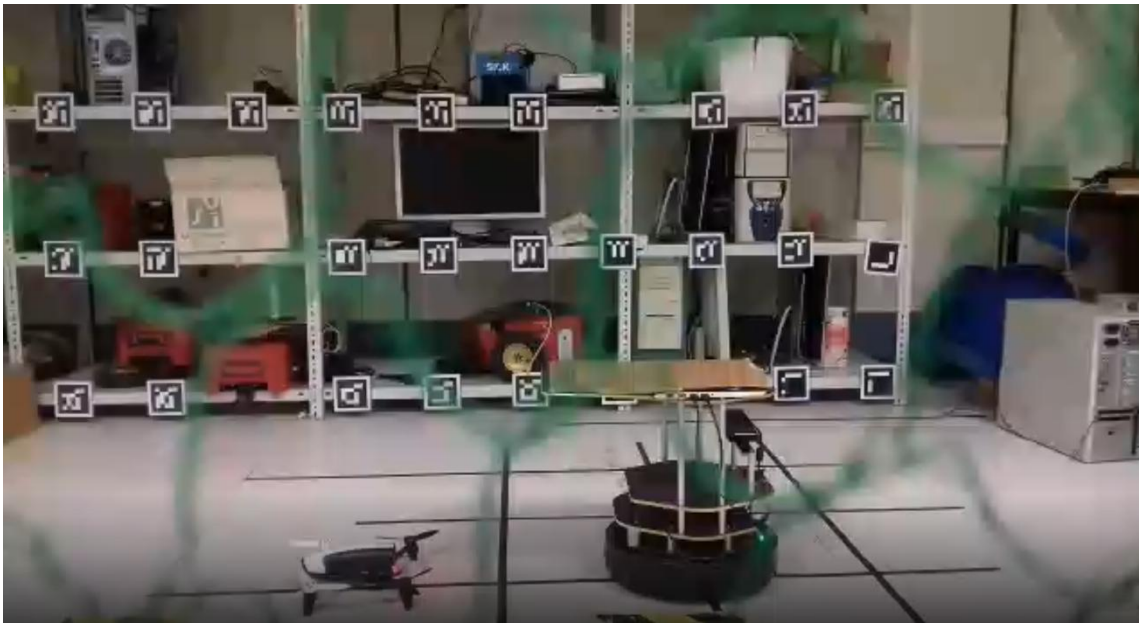

Figura 6 Disposición de los robots antes del inicio de la prueba

Para este ensayo se utilizaron todos los componentes descritos con anterioridad, la disposición de los mismos es la que viene dada por la figura 6. Se realizaron varios ensayos con anterioridad con el UAV en otra posición, sin embargo, se posicionó delante al estar más cerca éste de la webcam del ordenador, con el objetivo de ahorrar batería.

La colocación y tamaño de la etiqueta necesaria para la detección del dron es de vital importancia, ya que si es demasiado grande puede alterar el vuelo del UAV y si está próxima al sensor de altitud que lleva puede dar medidas erróneas y provocar problemas. Tal y como se muestra en la figura 7, la etiqueta está colocada en la parte inferior del dron en el lado opuesto a la cámara.

Figura 7 Imagen cámara USB

El ensayo consta de tres partes que se detallarán en los apartados siguientes.

4.2.1. Parte 1: Encendido general de los dispositivos

Para tenerlo todo operativo es necesario establecer qué es lo que utilizaremos y para qué. Así pues se puede dividir este encendido general en tres partes, ya que se trabajará principalmente en tres dispositivos.

El Turtlebot o UGV será en nuestro caso el dispositivo principal de los tres, solo por el simple hecho que desde él se iniciará el roscore y los demás se conectarán a él para poder establecer la conexión de paquetes que se utilizará. Además, se iniciarán los drivers del Turtlebot, (`turtlebot_bringup minimal.launch`) para poder controlar el UGV mediante el paquete `turtlebot_teleop`. Para poder ver al dron sobrevolar el Turtlebot habilitamos la cámara USB del portátil con el nodo `usb_cam_node` alojado en el paquete `usb_cam`. La figura 8 corresponde a la imagen recogida por la cámara USB durante el ensayo.

Hay que destacar que la información que nos proporciona el driver del Turtlebot es de vital importancia para saber tanto la posición del UGV con respecto a la posición inicial de encendido así como la orientación de éste en cuaterniones. Este hecho se comentará en las secciones posteriores, ya que es muy importante para la elaboración de mapas.

Aunque el UAV está siendo manipulado desde el centro de mando o PC de sobremesa, también es tratado como otro dispositivo a tener en cuenta, ya que hay paquetes que se utilizarán expresamente para él.

Para tenerlo activo, el primer paso es pulsar el botón posterior que parpadeará unos segundos y después se pondrá en un rojo sólido. Posteriormente, para habilitar todas las funcionalidades del Parrot, se iniciará el driver del mismo mediante un roslaunch como viene a continuación: `roslaunch bebop_tools bebop_nodelet_iv.launch`. La razón por la cual se ha iniciado el driver de esta forma y no de otra es porque de esta forma fue iniciada en el artículo en el cual se ha basado este proyecto. La única diferencia con la iniciación como nodo es que iniciándolo como nodelet también lo empieza todo en su espacio de nombre propio y carga la configuración por defecto.

Para teleoperar el dron, se utilizó el paquete genérico `teleop_twist_keyboard` el cual servirá para posicionar el UAV sobre el Turtlebot.

El ordenador de sobremesa, aunque no es el elemento principal es el elemento desde donde, vía ssh hacia el Turtlebot o vía terminal desde el propio PC, se iniciarán todos los procesos. Desde el PC se iniciará el nodo `individualMarkersNoKinect` del paquete `ar_track_alvar`, este nodo identificará una etiqueta con una cámara monocular. También se iniciará el nodo que posibilitará la cooperación entre el Turtlebot y el Parrot, el nodo `tracking_nodev2`. Este último nodo se explicará con detalle en apartados posteriores.

4.2.2. Parte 2: Aproximación de los robots

Una vez que todo está dispuesto, es necesario seguir los pasos que se detallarán a continuación, pero antes de que eso suceda, es necesario ver las imágenes que está capturando la cámara USB situada encima del Turtlebot. Para ello, ejecutamos el nodo `image_view` enlazando el *topic* subscriptor con el *topic* publicador de la imagen de la cámara. De esta forma podremos ver en una ventana las imágenes que ésta captura.

Dicho lo anterior, la fase de aproximación consta de dos pasos bastante simples:

1.- Ambos, tanto Parrot como Turtlebot deben situarse a una distancia no inferior a los 20 cm con respecto del otro por motivos de seguridad. Ya que durante el despegue del dron es posible que éste pueda realizar un movimiento que provoque la colisión con el UGV. Es muy recomendable situar ambos robots en un entorno despejado y con redes de seguridad.

2.- Una vez confirmado que todo se ha iniciado correctamente, se procederá al despegue del dron por medio del comando (`rostopic pub --once bebop/takeoff std_msgs/Empty`), y se pilotará con el nodo `teleop_twist_keyboard` hasta estar emplazado encima de la cámara USB.

4.2.3. Parte 3: Movimiento conjunto

Una vez todo en posición, se ejecuta el nodo desarrollado, `tracking_nodev2`, el cual se describirá posteriormente. Se realizarán movimientos simples hacia adelante y hacia atrás, además de giros. Los movimientos elegidos son denominados simples, es decir, son traslaciones donde una sola componente es distinta de cero.

Descritas las etapas en detalle, solo queda ver cómo funcionan los componentes y los resultados que se han obtenido a partir de su uso.

4.3. Funcionamiento del nodo `tracking_nodev2`

El nodo se ha realizado en Python [12] por su simplicidad para elaborar nodos de ROS en lugar de utilizar C++.

Para comprender como funciona el programa desarrollado `tracking_nodev2`, es necesario saber al menos qué es lo que publica el nodo `individualMarkersNoKinect`, el cual es el encargado de reconocer la etiqueta y determinar su posición y orientación.

Este nodo publica mensajes en el *topic* `/ar_pose_marker` del tipo AlvarMarker, el cual posee la estructura que se puede apreciar en la figura 8.

```
header:
  seq: 0
  stamp:
    secs: 1444430928
    nsecs: 28760322
  frame_id: /head_camera
id: 3
confidence: 0
pose:
  header:
    seq: 0
    stamp:
      secs: 0
      nsecs: 0
    frame_id: ''
  pose:
    position:
      x: 0.196624979223
      y: -0.238047436646
      z: 1.16247606451
    orientation:
      x: 0.970435431848
      y: 0.00196992162831
      z: -0.126455066154
      w: -0.205573121457
```

Figura 8 Estructura de mensaje AlvarMarker

El nodo `tracking_nodev2` se subscribe al *topic* `/ar_pose_marker` y accede a la pose de los mensajes publicados, permitiendo conocer los valores de posición y orientación. La misión del nodo es conseguir que el dron esté en un área concreta con respecto a la cámara, por lo que tratará de ir componente a componente publicando mensajes sobre el *topic* `/bebop/cmd_vel` en caso de que los valores no estén dentro de los valores deseados.

Como bien se ha visto antes, los ejes de la etiqueta Alvar no coinciden con los ejes que posee el dron, por lo que el nodo, al evaluar el valor de la coordenada de la etiqueta, tendrá que mandar un comando de velocidad lineal a x si se pretende mover el dron a lo largo del eje y, o mandar un comando de velocidad lineal a y, si se desea desplazar el UAV por el eje x.

Para incrementar la altura del dron o disminuirla, se publica un comando Twist que solo posea un valor no nulo en la componente lineal z. En este caso, la altura del dron incrementará para valores positivos y disminuirá para valores negativos.

También se realiza un control sobre la orientación del Parrot, aunque se prioriza que el Parrot esté centrado.

Una vez que el UAV esté en posición, se evaluará la orientación de la etiqueta localizada en éste. Como el valor viene dado en cuaterniones, se ha investigado mediante el uso del paquete `ar_track_alvar`, cuál es la componente que interesa para hacer girar el dron de la manera que realmente se desea.

Para cambiar la orientación del UAV se debe observar el valor de la componente x de la orientación del mensaje AlvarMarker recibido. Si es mayor de 0, se dará un comando de velocidad angular no nulo en z que sea negativo, y al contrario si el valor de x es menor de 0.

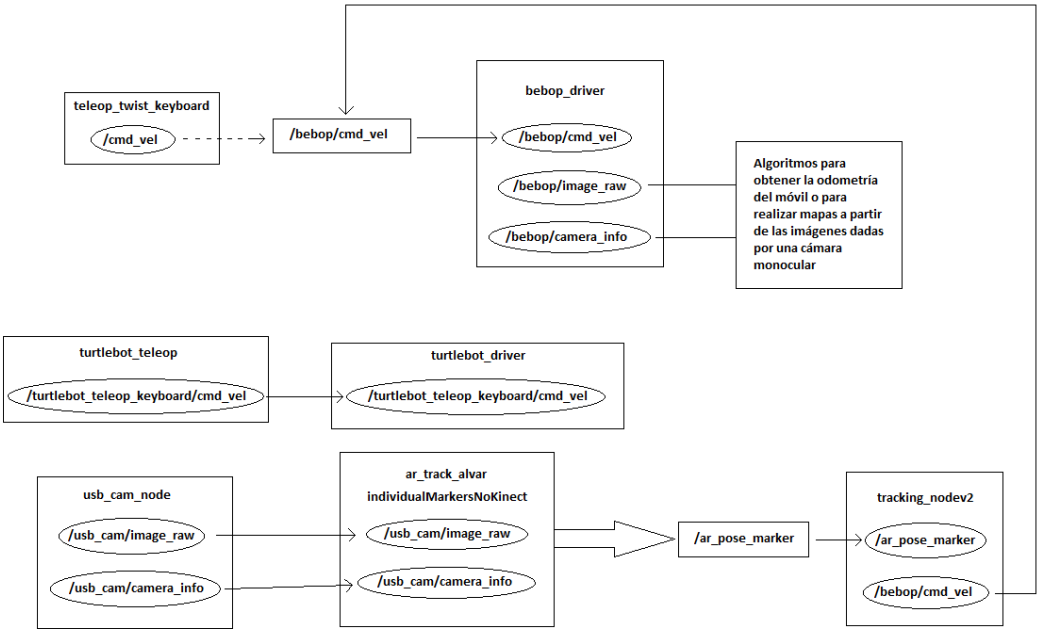

Figura 9 Esquema simplificado de los *topics*

La figura anterior (figura 9) es una versión simplificada del mapa de *topics* y nodos que se ha generado, recogiendo solo los que han intervenido en el ensayo.

Como podemos ver en el esquema, están los drivers o controladores del Turtlebot y Parrot en los cuales destacan los *topics* en los que se publican los comandos Twist de velocidad para realizar su movimiento.

También se puede ver que el nodo `teleop_twist_keyboard` posee una línea discontinua que va desde el *topic* `/cmd_vel` hasta `/bebop/cmd_vel`, lo que quiere decir que la comunicación entre un nodo por medio del *topic* solo está disponible en la fase de aproximación del dron hacia la posición del Turtlebot.

Para controlar el Turtlebot se ha utilizado el nodo `turtlebot_teleop`, el cual viene preinstalado en el propio UGV y no necesita renombrar el *topic*, como ocurría en el caso anterior para controlar el dron.

Como bien se ha explicado antes y se ha representado en la figura anterior, `ar_track_alvar` se sirve de las imágenes suministradas por el nodo de la cámara `usb_cam_node` y por la calibración de ésta, publicada en `/usb_cam/camera_info`.

En el esquema anterior se ve que el nodo `tracking_nodev2` recibe los mensajes Alvar del *topic* `/ar_pose_marker` y manda los comandos Twist al *topic* `/bebop/cmd_vel`.

4.4. Estudio previo y justificación

Para entender por qué se ha llegado a esta solución es necesario explicar cómo se ha tratado de abordar el problema desde un inicio.

Al comienzo, se investigó sobre múltiples formas de cooperación entre robots, llegando a la que inspiró el desarrollo de este trabajo, *Bird's Eye View*, el cual poseía componentes que de una forma o de otra se han tenido presente y se han ido descartando en el proceso con el objetivo de simplificar y llegar al núcleo del proyecto que era la cooperación entre robots.

De entre todos los proyectos que se investigaron, siempre se obtuvo un denominador común, la forma de saber que un robot estaba en una determinada posición venía dada por la segmentación de la imagen que recibía el otro robot o el uso de etiquetas sobre los mismos. De esta forma, se procedió a realizar el programa descrito con anterioridad, pensando en múltiples formas de realizar la cooperación y teniendo presente el artículo *Bird's Eye View*, se utilizó el paquete `ar_track_alvar` para hacer que los robots se movieran en conjunto.

Al inicio, se pensaba utilizar una idea similar a la que consideraron en el artículo mencionado con anterioridad, que era la de utilizar dos etiquetas, una que estuviera situada en el dron, y otra que estuviera al lado de la cámara USB. El objetivo de poseer dos etiquetas era obtener robustez frente a las perturbaciones a las que el Parrot pudiera estar siendo sometido. Como se apreciará en las figuras de apartados posteriores, además de que los mensajes de `ar_pose_marker` poseen un ruido bastante alto, el dron para una posición fija necesitaba ser reconducido por el nodo creado, es decir, la posición que se suponía que era de reposo no lo era.

Ante todo esto, había varias alternativas a considerar para realizar el control del dron, entre las cuales estaban las siguientes:

1.- Realizar un control PID de posición tal y como lo habían hecho en *Bird's Eye View*. El cual consistía en centrar el Parrot en (0,0,45) cm. Esta alternativa fue descartada debido a exigencias temporales, además de la complejidad que supone hallar las constantes del PID para que realice movimientos compuestos.

2.- Realizar un control PID en posición para cada componente de posición y orientación de forma que se diesen órdenes donde solo una componente de velocidad fuese superior a cero. Se perseguiría que el dron estuviera en la posición (0,0,45) cm. Esta alternativa fue rápidamente descartada, por la complejidad que conllevaba el hecho de realizar varios PID para distintas componentes, y como en la primera alternativa, también requería de un tiempo del que no se disponía para hallar las constantes de los controladores.

3.- En la línea de la alternativa anterior, también se contempló hacer actuar al PID si los valores estuvieran fuera de los márgenes que le fueran impuestos. Aunque esta alternativa pudiera ser una solución a nuestro problema, se decidió no implementar debido a la complejidad de la misma, teniéndola presente en el caso de que las exigencias temporales pudieran ser satisfechas con premura.

4.- Realizar un control simple que mande comandos cuando el Parrot esté fuera de los límites especificados. Las componentes de velocidad del comando serán todos nulos exceptuando uno a uno de ellos, el cual tratará de corregir la componente que se haya salido de los márgenes. Se eligió esta opción debido a su simpleza y con vistas a establecer una base de la que se pudiera partir para poder mejorar el trabajo.

-Problemas

A lo largo de la consecución del trabajo, múltiples fueron los problemas que fueron surgiendo, casi todos ellos relacionados con el desconocimiento. Estos problemas se explicarán a continuación.

-Problemas en la investigación

Como se ha dicho con anterioridad, fueron varios los problemas a los que se tuvo que hacer frente. En el caso de la investigación también los hubo. Cuando se buscan artículos relacionados a lo que se está investigando es normal que algunos no dejen del todo claro qué tipo de solución han utilizado finalmente. Además, las soluciones propuestas y dejadas en los repositorios de Github no siempre están del todo especificadas y aun habiendo sido testadas pueden contener errores sintácticos graves, por lo que es conveniente estar alerta cuando se revisa el código de un artículo.

Una de las razones anteriormente no nombradas de por qué se está revisando el código de los artículos de investigación, es para evaluar la repetitividad de los resultados y dictar si es la solución adecuada al problema que perseguimos resolver.

Otra de las razones para escudriñar el código es para obtener algún tipo de idea o basarse en él. Sin embargo, en este caso, cuando se ha analizado había ciertas líneas que no cuadraban con lo que realmente hacían las componentes que se utilizaban y aumentaban la confusión.

Finalmente, se decidió establecer que el dron siguiese al Turtlebot mediante el reconocimiento de una etiqueta que estuviera localizada en el Parrot. De esta forma, el proyecto se centraría en hacer que ambos robots mantuvieran la posición mientras navegan por el entorno.

-Problemas en la elaboración del ensayo.

Una vez se investigaron las distintas formas de afrontar el proyecto, se estableció un objetivo simple, hacer que el dron siguiera al Turtlebot mediante la etiqueta localizada en el dron. La idea era similar, como se ha dicho anteriormente, a como se había planteado en *Bird's Eye View*. El mayor inconveniente en esta fase del proyecto era el desconocimiento de como se debía proceder a volar el dron con la etiqueta puesta.

Tras varios aterrizajes forzosos, se investigó al respecto y se llegó a la conclusión de que el Parrot poseía un sensor de altura en la parte inferior que estaba siendo tapado por la etiqueta que servía para que el Turtlebot lo localizase, lo cual hacía que el dron creyera que estaba volando a una altitud bastante inferior, tratando de compensar este hecho, aceleraba inexplicablemente sus rotores y viajando hasta las redes de protección.

Una vez que se probó poniendo la etiqueta en otra parte que no perturbase al sensor de altitud, el principal problema que se encontró para sacar datos fue la duración de la batería. La cual el único remedio que tenía era poseer varias baterías a mano para poder operar durante mayor tiempo recogiendo información.

- El problema de los mapas

Uno de los mayores inconvenientes que se han afrontado es la elaboración de un mapa conjunto entre dron y Turtlebot. Una vez se ha realizado la meta de la cooperación entre los dos robots, las posibilidades pueden ser bastantes, desde hacer un mapa tridimensional hasta hacer que estos dos naveguen por un espacio conocido e intenten interaccionar con los objetos, ya sea mediante etiquetas o colores. Sin embargo, como se detallará posteriormente, solo se ha podido investigar cómo elaborar los mapas y cómo hacer distintas operaciones pero no se han podido poner en marcha, ya sea por motivos de seguridad o por problemas temporales.

4.5. Resultados y análisis posterior

Como bien se ha mencionado con anterioridad, se realizaron varios ensayos, todos ellos con la finalidad de calibrar y probar el programa realizado.

Teniendo todo en posición y correctamente funcionando, se pone en marcha el nodo.

Como este nodo no posee un PID o PD que regule la velocidad que se le dará al Parrot para que realice su movimiento, es necesario ir probando con distintas velocidades que puedan cumplir con el cometido, que es situar el dron dentro de los márgenes deseados.

Se empezó desde una velocidad lineal inicial de 0.09 m/s tanto para el Turtlebot como para el Parrot y se subió de a escalones de 0.005 m/s. Lamentablemente, cuando el dron superaba los 0.1 m/s, se perdía y había que redirigirlo con el paquete `twist_keyboard_teleop` hasta la posición deseada. Por lo que se usó como máximo impulsos de valor 0.1 m/s para las velocidades lineales.

Para la velocidad angular se utilizó desde un inicio valores inferiores a 0.5 rad/s, el resultado de éstos fue que la rotación era demasiado lenta. En cambio para valores superiores a 0.5 rad/s la rotación era demasiado brusca y no se alcanzaba la posición deseada de forma efectiva.

Hay que recordar que este nodo está centrado en hacer que la posición del dron esté dentro de unos márgenes, la orientación de éste casi se deja en un segundo plano, aunque se realiza, como bien se apreciará en las siguientes gráficas.

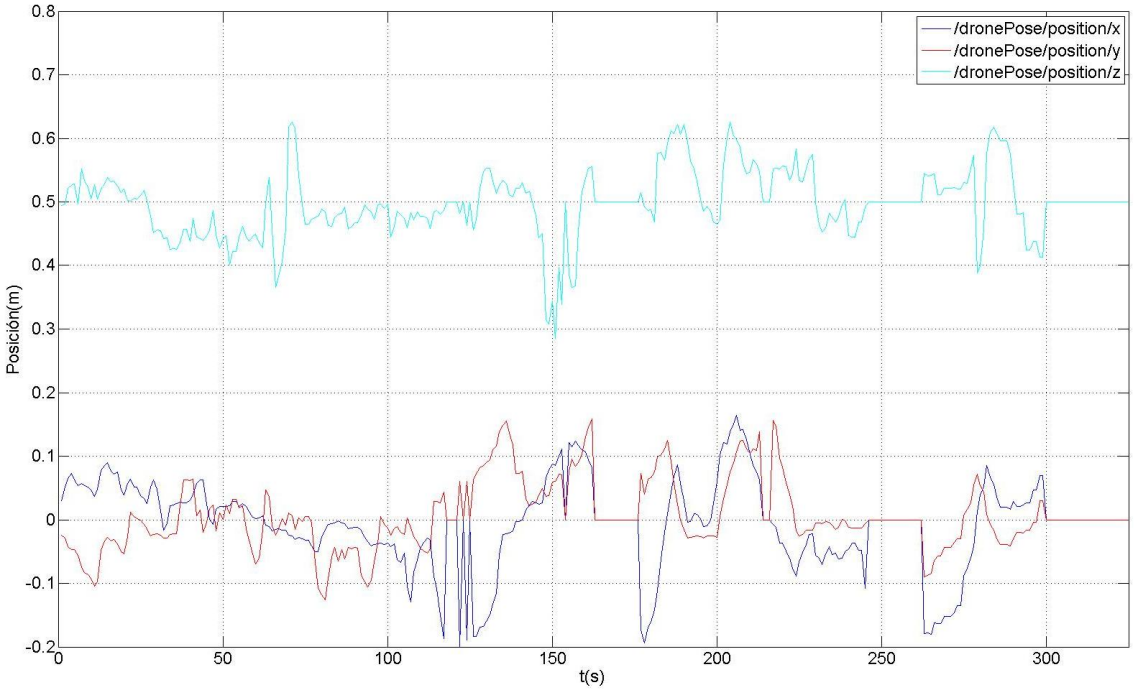

Figura 10 Posición del dron durante la prueba

Todas la gráficas mostradas son sacadas por el paquete `rqt_plot` de ROS. Hay que añadir que la posición y orientación de la tarjeta se han retransmitido por el nodo `tracking_nodev2`, ya que sucedía un problema cuando no se podía ver la etiqueta.

La figura 10 muestra los valores de (x, y, z), en metros, de la etiqueta localizada en el dron con respecto un eje de abscisas (eje x) que se incrementa debido a los mensajes que son publicados en el *topic* `/dronePose`. La gráfica anterior y las siguientes recogen el movimiento del dron y los comandos dados al UAV en una prueba de aproximadamente 7 minutos y 30 segundos en los que tanto el dron como el Turtlebot tratan de realizar un movimiento conjunto. Tal desplazamiento consta de un primer tramo en el que el dron trata de ser centrado mediante el nodo programado y explicado en este trabajo. Esta fase dura aproximadamente 1 minuto y 15 segundos.

El siguiente tramo de tiempo corresponderá a un desplazamiento hacia adelante que durará 20 segundos y después se realizará otro movimiento hacia atrás que durará aproximadamente 50 segundos.

Hacia el minuto 2:28 de la prueba, se hará un giro hacia la izquierda. Y se alternarán los movimientos mencionados anteriormente con diferentes duraciones cada uno con el objetivo de probar la robustez del programa realizado.

Todos estos movimientos se realizaron mediante comandos dados al Turtlebot de valor 0.09 m/s para las componentes lineales, y 0.5 rad/s para las componentes angulares. Los mismo valores recibió el dron a través del nodo `tracking_nodev2`.

Como se puede ver en la gráfica anterior y en las siguientes, hay zonas en las se perdía la visión de la etiqueta. Cuando eso sucede el nodo manda una posición fija, de ésta manera el dron no recibirá ningún comando de movimiento. La posición que manda es (0.0, 0.0, 0.5, 0.0, 0.0, 0.0),

donde los tres primeros componentes corresponden a la posición del móvil y los tres últimos a la orientación del mismo.

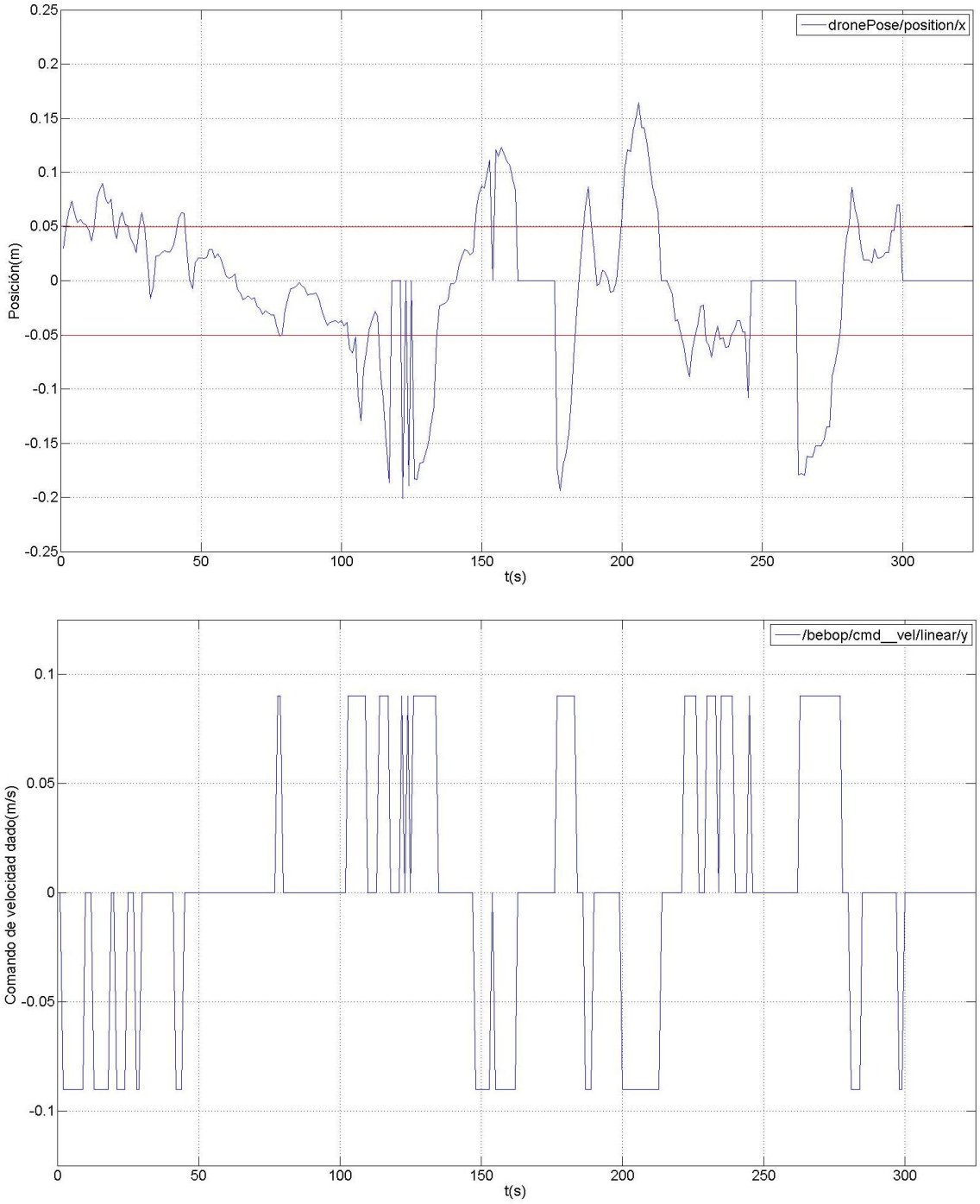

Figura 11 Posición x del dron (etiqueta) durante la prueba, y valores de velocidad "y" dados para mantener la posición x dentro de los márgenes

Como bien se ha explicado con anterioridad, `ar_track_alvar` posee unos ejes distintos a los que utiliza el Parrot para realizar su movimiento, por lo que éste hecho debe ser reflejado en el programa. En la gráfica anterior (figura 11) vemos como para controlar la posición x del dron se tiene que utilizar la componente y de la velocidad lineal.

También podemos apreciar que posee una cantidad de perturbaciones bastante alta. El programa trata de corregir en todo momento la posiciones de forma que el UAV siempre esté situado dentro de los márgenes dados, que estarán entorno a 0.05 metros.

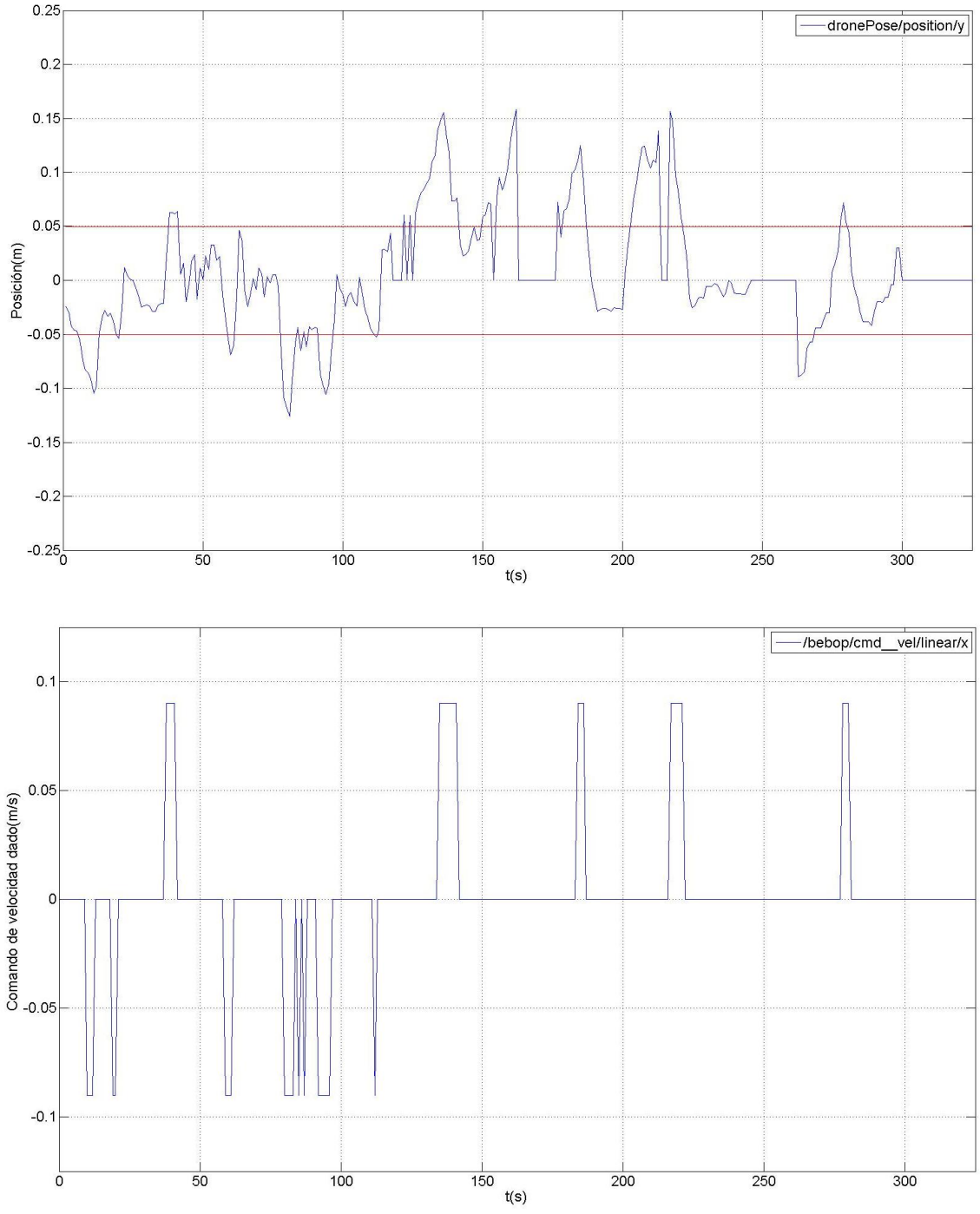

Figura 12 Posición y del dron (etiqueta) durante la prueba, y valores de velocidad x dados para mantener la posición "y" dentro de los márgenes

La posición y del dron vendrá controlada por la componente x de la velocidad lineal. Ésta es la coordenada que se controla en primera instancia, como se puede apreciar en la gráfica (figura 12), los impulsos sitúan perfectamente dentro de los márgenes al UAV. Cabe destacar que, como se ha nombrado antes hay tramos donde la etiqueta se pierde. De vital importancia es el

tramo final, donde la etiqueta no es localizada, ya que hubo un fallo del sistema donde todos los programas quedaron bloqueados.

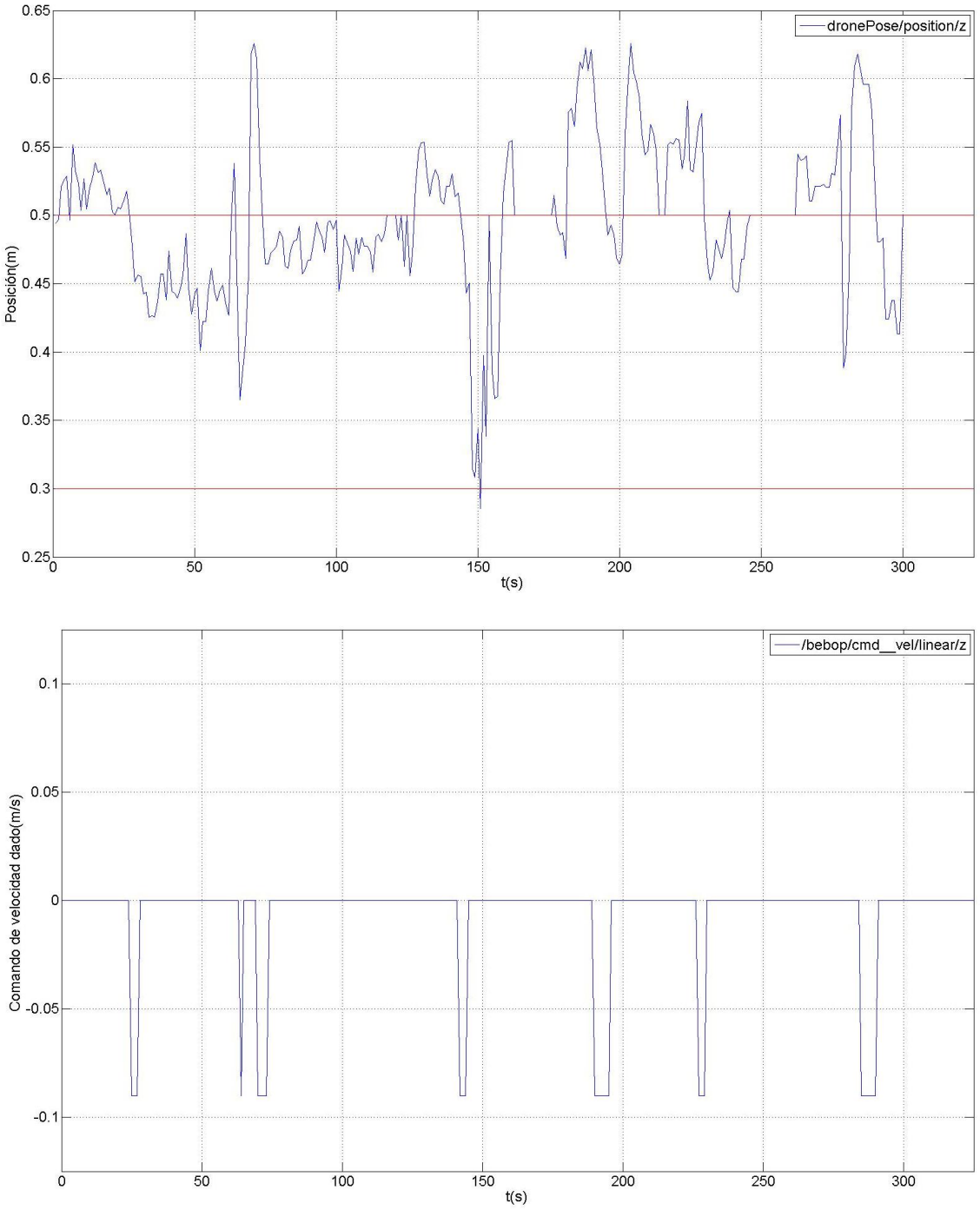

Figura 13 Posición z del dron (etiqueta) durante la prueba, y valores de velocidad lineal z dados para mantener la posición z dentro de los márgenes

Para la coordenada z o altura, se utilizó la componente z de la velocidad lineal.

A medida que el ensayo avanzaba, la altura del dron varia de forma bastante significativa, esto puede deberse en mayor parte a que el dron por motivos de seguridad estaba próximo al Turtlebot, por lo que el sensor determinaría que estaba cercano al suelo. Este hecho, haría que

el Parrot ascendiera de manera brusca en contra de nuestros deseos, que era permanecer como máximo a 0.5 metros con respecto al Turtlebot.

Como se aprecia en la gráfica (figura 13), hay varias subidas de altura y correcciones posteriores.

Para controlar el yaw del UAV se utiliza la componente z o yaw de la velocidad angular, como suele ser lógico en todos los dispositivos.

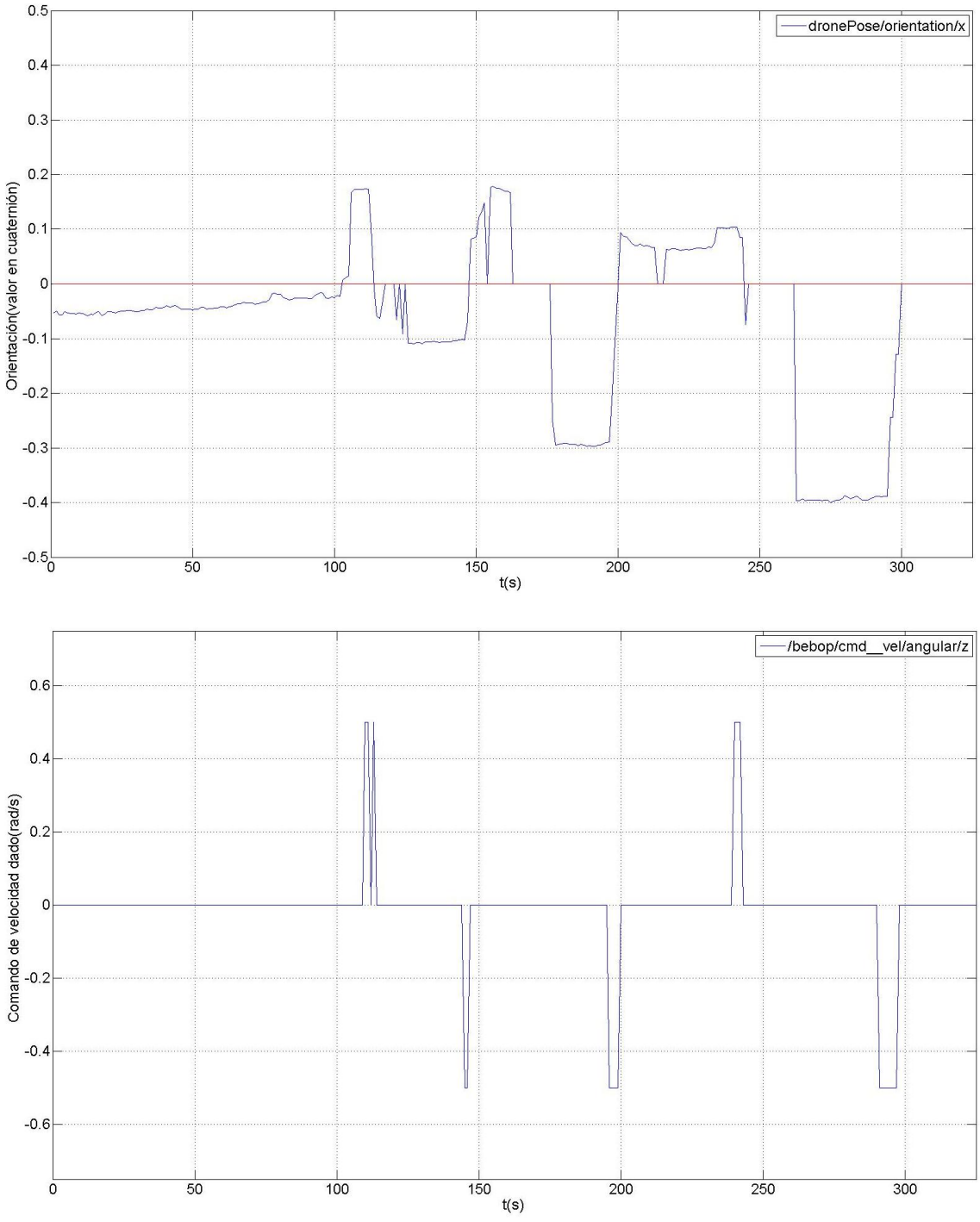

Figura 14 Orientación z del dron durante la prueba y valores de velocidad angular dados para mantenerlo en la referencia

Sin embargo, como se ha explicado con anterioridad, el yaw del dron se corresponde con el roll de que nos ofrece mediante los mensajes AlvarMarkers el paquete `ar_track_alvar`. El valor de ésta componente viene dado en cuaterniones.

Como se puede apreciar en la gráfica (figura 14), se trata de realizar un control sobre la orientación del dron, aunque como se ha dicho antes se prioriza la posición con respecto de la orientación.

Las demás componentes de la orientación no tienen por qué ser modificadas, ya que sirven para un propósito muy específico, el cual no interesa por el momento.

5. Conclusiones

Como bien se ha mencionado anteriormente, este trabajo está basado en un artículo [1], a partir de éste, se ha estado investigando el código propuesto por el mismo con la intención de replicarlo en la medida de lo posible. Tratando de ésta manera comprender cómo funcionaba y cuál era el sentido de la inclusión de los componentes de los que estaba hecho.

Este hecho a pesar de ser tedioso por tener que ver el código de otros desarrolladores, arrojó cierta luz sobre los nodos programados.

Al examinar el trabajo realizado en el repositorio de código se determinó realizar una aplicación sencilla en Python [12], debido a que había ciertas discrepancias entre los distintos componentes que conformaban el artículo [1].

La utilización de `ORBSLAM2` aunque era eficaz al ser utilizado por separado, no publicaba la posición estimada por éste en ROS. Sin embargo, en el nodo que realizaba la cooperación sí que establecía que había un *topic* del cual recogía la posición del dron y se hacían operaciones con ella, pero además se publicaba la posición del dron obtenida a partir de la posición de la etiqueta AR situada en éste con respecto de la cámara del robot móvil terrestre. Todo esto motivó la realización del nodo anteriormente explicado, el cual es mucho más simple que el programado en [1] pero que cumple una función similar.

Por lo tanto cabe destacar que comparar ambos métodos puede ser bastante difícil, ya que en cuanto a recursos utilizados, el explicado en este documento es el más liviano, además de establecer una base sólida para posibles ampliaciones que se comentarán más adelante.

No obstante, al realizar solo la cooperación sin determinar la posición del dron en el entorno, convierten al menos al código presentado en el artículo en un trabajo más completo, a pesar de que posee las incongruencias anteriormente mencionadas.

Por lo que a pesar de las comparaciones que se han realizado con brevedad, no es posible establecer cuál de las dos soluciones presentadas es la mejor.

Aún así, sí que es posible determinar cuáles son los puntos débiles de este trabajo cuáles son sus puntos fuertes.

Puntos fuertes

- Es un programa simple que tramita mensajes AlvarMarkers y da comandos de velocidad al dron a partir del contenido de los mensajes.
- Está totalmente integrado en ROS, es decir, todos los programas utilizados son reconocidos por ROS y además son bastante conocidos por los desarrolladores de ROS, por lo que cualquier duda que se tenga puede ser resuelta con una simple búsqueda.

Puntos débiles

- El principal punto débil de este trabajo, es que aunque realiza la cooperación, no realiza ninguna tarea más allá del movimiento conjunto, por lo cual aún posee un margen muy alto de mejora.
- No utiliza controladores para posicionar el dron en la posición deseada.

6. Trabajo futuro

Como se ha mencionado con anterioridad, este trabajo es muy mejorable. Ya que la piedra angular está posicionada en esta sección se comentarán diferentes mejoras a tener en cuenta para extender las funciones del conjunto.

En el primer momento, se trató de hacer que ambos robots cooperaran para tratar de realizar un mapa tridimensional del entorno en el cual estuvieran emplazados, sin embargo, como se ha mencionado antes, utilizaba elementos como `ORBSLAM2` [8] el cual no estaba totalmente integrado con ROS [4]. Por lo tanto, sería necesario encontrar un paquete que pudiera realizar la labor de `ORBSLAM2` y que estuviera totalmente integrado en ROS [4] o en su defecto encontrar la forma de estimar la odometría del dron a partir de las imágenes publicadas. Para ambas alternativas hay solución.

Posibles soluciones

Como bien se ha comentado antes, uno de los problemas que se tienen que resolver es la estimación de la odometría del dron. Para ello se tienen varias alternativas:

- Se puede estimar la odometría de manera similar a una de las formas que utilizan en el artículo, la cual utiliza la posición del Turtlebot y la posición de la etiqueta Alvar localizada en el dron. Sumando cada coordenada de posición obtenemos la posición del dron y su recorrido. Sin embargo, esta estimación, al igual que en artículo necesita un respaldo que le dé robustez frente a posibles ruidos.
- Para estimar la posición del dron, se puede utilizar el paquete `viso2_ros` [10], el cual necesita del paquete `image_proc` [11], que proporciona las imágenes rectificadas, para funcionar. Además, es posible que necesite de un coeficiente de escala para la obtención de las medidas. Es una opción viable, pero que requiere tiempo. Esta opción complementaria a la anterior para la obtención de la estimación.
- Otra de las alternativas es utilizar el paquete `rpg_svo`, que viene descrito en el artículo Fast Semi-Direct Monocular Visual Odometry [5]. Es una solución que tal y como dicen es robusta, rápida y precisa. Lo más importante es que haría el papel de `ORBSLAM2` en el artículo [1]. Esta alternativa no sería complementaria de las anteriores, pues realiza el trabajo de manera eficiente por sí sola. El único requisito para poder utilizar este paquete es que la versión de ROS debe ser Indigo o inferior.
- Por último, de entre otras muchas opciones y para versiones superiores a ROS indigo tenemos el paquete `VINS-Mono` descrito en el artículo [6] mucho más actual al anterior. Y que realiza el mapeado a partir de las imágenes recibidas por la cámara monocular.

Vistas las soluciones para estimar la odometría del dron y mapeado realizado por el mismo, se podría realizar el mapeado conjunto entre los dos robots, pues Turtlebot posee un archivo .launch (`gmapping_demo.launch`) de serie que elabora el mapa bidimensional a partir de los datos recibidos por sus encoders e imágenes del sensor Orbbec Astra. Sería cuestión pues de establecer la correspondencia de los dos mapas para elaborar un mapa tridimensional con un error reducido.

Referencias

[1] Shannon Hood, Kelly Benson, Patrick Hamod, Daniel Madison, Jason M. O'Kane, and Ioannis Rekleitis, Bird's Eye View: Cooperative Exploration by UGV and UAV, 2017 International Conference on Unmanned Aircraft Systems (ICUAS)

[2] Open Source Robotics Foundation, Inc., TurtleBot, https://www.turtlebot.com/

[3] Parrot SA, Parrot, https://www.parrot.com/es/drones/parrot-bebop-2#t%C3%A9cnicos

[4] Enrique Fernández et al., Learning ROS for Robotics Programming - Second Edition, August 2015

[5] Christian Forster, Matia Pizzoli, Davide Scaramuzza, Fast Semi-Direct Monocular Visual Odometry, IEEE International Conference on Robotics and Automation (ICRA), 2014

[6] Tong Qin, Peiliang Li, Zhenfei Yang, Shaojie Shen, VINS-Mono: A Robust and Versatile Monocular Visual-Inertial State Estimator, IEEE Transactions on Robotics 2018, 34, 4, 1004-1020

[7] Scott Niekum, ,ar_track_alvar, Recuperado de https://github.com/ros-perception/ar_track_alvar.git

[8] Raúl Mur-Artal, Juan D.Tardos, J. M. M. Montiel, Dorian Galvez-Lopez, ORBSLAM: a Versatile and Accurate Monocular SLAM System, IEEE Transactions on Robotics 2015, 31, 5, 1147-1163

[9] Vincent Rabaud, camera_calibration, Recuperado de https://github.com/ros-perception/image_pipeline.git

[10] Stephan Wirth, viso2_ros, Recuperado de https://github.com/srv/viso2.git

[11] Vincent Rabaud, image_proc, Recuperado de https://github.com/ros-perception/image_pipeline.git

[12] Allen B. Downey, Think Python, 2012

[13] VTT Technical Research Centre of Finland Ltd, ALVAR, http://virtual.vtt.fi/virtual/proj2/multimedia/alvar/index.html